Comment on Software Efficiency: Loops, Subroutines, and Interpretive Execution

J. W. Layland
Communications Systems Research Section

This article will discuss the relationship between the efficiency of software operation and the use within that software of conventional control structures such as loops and subroutines. These control structures 'fold' a program to reduce its storage requirements at the expense of increased execution time. The typical intuitive response to this consideration is one of "the faster, the better." In some contexts, this is far from correct. The extent to which folding should be done depends upon many factors, including especially the program's total size. The extent to which this folding can actually be done depends upon details of its operation, so the analysis presented in this article will of necessity retreat from reality, and assume that the program in question can be folded arbitrarily.

I. Introduction

This article will discuss the relationship between the efficiency of software operation and the use within that software of conventional control structures such as loops and subroutines. These control structures 'fold' a program to reduce its storage requirements at the expense of increased execution time. The typical intuitive response to this consideration is one of "the faster, the better." It will be seen shortly that in some contexts, this is far from correct. The extent to which folding should be done depends upon many factors, including especially the program's total size. The extent to which this folding can actually be done depends upon details of its operation, so the analysis presented in this article will of necessity retreat from reality, and assume that the program in question can be folded arbitrarily. We assume further that we are in a position to be able to pay for only as much storage and computing time as actually used. This implies either that we are using some small part of a fairly priced multiprogrammed computation center, or that we are assembling a computational facility that is being optimized for our particular task.

II. Problem Definition and Notation

Assume that a task has been defined in terms of a *Straight Line Machine Algorithm* (SLMA) of S steps on a standard computational resource. The label, SLMA, is a

"corruption" of the logical straight line algorithms considered by Savage (Ref. 1), and implies only that each step is used exactly once in task execution, and no loops or other control strategies are employed that cause multiple usage of steps. No specific machine implementation is implied. It is the cost of various implementations of this task that is of interest. Were we, for example, to implement the task directly in the SLMA form, it would take S units of time, and S storage elements, resulting in a cost that is proportional to S^2 for the storage of the algorithm alone. This is not the only cost element in the machine. Other machine costs, denoted in total as M for the remainder of this article, will be assumed to be normalized by the cost of a single storage element. M consists of M_c, computer processor cost; M_d, task data cost; and M_p, external, or peripheral cost. For a medium-scale processor, M_c is currently on the order of 10^4 or larger. Common minicomputers have an M_c equal in cost to about 10^3 instruction storage elements, and the integrated circuit micro-computers have, or soon will have, an M_c below 10^2. The total cost of the directly implemented SLMA is

$$C_1 = S^2 + S \cdot M \tag{1}$$

The first term of Eq. (1) represents cost of storing the algorithm in a computer memory, and can be drastically reduced by use of the control structures of interest. At the same time, the execution time is increased by the overhead needed to perform the control function. For the remainder of this article, θ will be used to denote the amount of overhead, in machine steps, needed to perform one control operation in a folded implementation of an algorithm.

III. Loops

A loop exists in an algorithm implementation whenever some particular sequence of steps of the algorithm is executed repeatedly. This sequence of steps is followed by the θ overhead steps that determine whether the algorithm steps should be executed again, that modify pointers to the data manipulated by the algorithm, and do other house-keeping chores that are not inherent to the algorithm, but that are needed because of the way it is implemented.

Suppose that our task of S steps can be folded arbitrarily, and that it is to be implemented by repeated executions of a loop of β instructions. This loop must obviously be executed S/β times. Including overhead, the total time is $(\beta + \theta) \cdot S/\beta$, and the total storage is $\beta + \theta + M$. The total operating cost is

$$C_2 = (\beta + \theta + M) \cdot (\beta + \theta) \cdot S/\beta \tag{2}$$

We wish to determine a value for β such that Eq. (2) is minimized. For $M \approx 0$, the optimum β can be determined trivially by treating β as a continuous variable; differentiating by it, and solving the resultant equation. The optimum β is equal to the overhead θ. At this point ($M = 0$), the minimum task cost, C_2^* is

$$C_2^* = 4\theta \cdot S \tag{3}$$

which is a linear function of S.

For nonzero M, the optimum β is again approximately determined by treating β as continuous; differentiating by it, and solving

$$0 = \frac{\partial C_2}{\partial \beta} = \frac{S}{\beta}(\beta + \theta)(\beta + \theta + M) \times \left[\frac{-1}{\beta} + \frac{1}{\beta + \theta} + \frac{1}{M + \beta + \theta}\right] \tag{4}$$

To solve Eq. (4), make the change of variables $\eta = M/\theta$, and clear fractions, producing

$$[\beta + \theta(1 + \eta)](\beta + \theta) = \beta[\beta + \theta(1 + \eta)] + \beta(\beta + \theta) \tag{5}$$

After eliminating common terms from Eq. (5), its solution can be seen to be

$$\beta^* = \theta(1 + \eta)^{1/2}$$
$$C_2^* = S \cdot \theta \cdot [1 + (1 + \eta)^{1/2}]^2 \tag{6}$$

Suppose that instead of being able to fold the algorithm into one loop, we were forced to use several, say N, with the i-th loop containing β_i steps. Each of the loops is executed for S/N total steps, causing $S/(\beta_i \cdot N)$ repetitions of the i-th loop. Hence total task time is

$$T = \sum_{i=1}^{N} \frac{S}{N} \cdot \frac{1}{\beta_i}(\theta + \beta_i)$$

and the storage required is

$$W = M + \sum_{i=1}^{N} (\theta + \beta_i)$$

since the overhead steps are required for each of the loops. The task operational cost is

$$C_3 = \left[M + \sum_{i=1}^{N} (\theta + \beta_i) \right] \cdot \left[\sum_{i=1}^{N} \frac{S}{N} \cdot \frac{1}{\beta_i} (\theta + \beta_i) \right] \tag{7}$$

By symmetry, $\beta_i = \beta_0$ for all i, so that

$$C_3 = \left[\frac{M}{N} + \theta + \beta_0 \right] \cdot S \cdot N \cdot \left[\frac{1}{\beta_0} \right] \cdot [\theta + \beta_0] \tag{8}$$

The solution to Eq. (8) can be determined by analogy with Eqs. (2) and (6) above. Let $\xi = M/(N \cdot \theta)$. Then

$$\begin{aligned} \beta_0^* &= \theta\,(1 + \xi)^{1/2} \\ C_3^* &= N \cdot S \cdot \theta \cdot [1 + (1 + \xi)^{1/2}]^2 \end{aligned} \tag{9}$$

For moderate M, the minimum C_3 represents an N-fold increase in cost over the minimum C_2. For large M, the minimum of both C_2 and C_3 approaches $S \cdot M$.

In an actual program, one would never have the option to fold a program as arbitrarily and completely as was done here. The option does exist to fold or not fold some particular sequence of steps, or perhaps to fold it partially. Suppose that the sequence of operations of current interest contains s_k steps that are to be folded (perhaps) into repeated execution of a loop of β_k steps. In Eq. (9), $1/N$ could be interpreted as the fraction of total algorithm steps performed by each of the loops. As such, $1/N$ corresponds to s_k/S in the present problem, and to a first-order approximation, the optimum β_k is given by

$$\beta_k^* = \theta \cdot \left(1 + \frac{M}{\theta} \cdot \frac{s_k}{S} \right)^{1/2} \tag{10}$$

If one of the allowed values for β_k falls close to this figure, then construction of the k-th loop at this figure is one portion of a minimum cost implementation. If allowed β_k values are below β_k^*, the s_k steps should almost certainly be implemented without the loop folding, because folding overhead adds proportionately more to task time than is saved in storage.

IV. Subroutines

Subroutines are a flexible means of causing multiple usage of specific steps within an algorithm implementation when these steps exist at scattered places within the algorithm. Just as with loops, a certain number of overhead steps must be executed to perform control operations of the subroutine linkage, and to establish data pointers for each distinct execution of the subroutine. As before, θ will denote the number of overhead steps per subroutine execution.

Suppose that our task of S steps can be folded arbitrarily, and that it is to be implemented by a sequence of calls to a subroutine of β instructions. There must obviously be S/β calls to this subroutine. Including overhead, the total time is $(\beta + \theta) \cdot S/\beta$, and the total storage is S/β for subroutine calls, $\beta + \theta$ for the subroutine, plus M for data, etc. The total operating cost is

$$C_4 = (M + \beta + \theta + S/\beta) \cdot (\beta + \theta) \cdot S/\beta \tag{11}$$

As before, we pretend that β is a continuous variable and locate the minimum of C_4 by differentiating with respect to β and solving for the zeros of that derivative. After some manipulation, we find that the minimizing β are the solutions to the cubic equation

$$\beta^3 = \beta\,(S + M\theta + \theta^2) + 2S\theta \tag{12}$$

For large S, the solution to Eq. (12) is approximately given by

$$\begin{aligned} \beta^* &\approx (S + M\theta)^{1/2} \\ C_4^* &\approx S \left[M + \frac{[2S + M \cdot \theta]}{(S + M\theta)^{1/2}} \right] \end{aligned} \tag{13}$$

The minimum cost is asymptotic to $2 \cdot S^{3/2}$ if $M << S$.

Subroutine structure is of course not limited to a single level. Subroutines can call subroutines that in turn call further subroutines ad infinitum. We would like to know how many levels can be used as well as how big each of the levels should be. Let β_i be the number of algorithm steps performed by one call to the i-th level subroutine, and let n be the number of subroutine levels. The n-th level subroutine then consists of β_n directly executed algorithm steps, and the i-th level subroutine (for $i \neq n$) consists of β_i/β_{i+1} calls to the $(i+1)^{st}$ level subroutine. Let $T(\beta_j)$ be the time to execute the steps at or below the β_j level. We consider $S = \beta_0$, $\beta_{n+1} = 1$. Then

$$\begin{aligned} T(S) &= \frac{S}{\beta_1} (\theta + T(\beta_1)) = \frac{S}{\beta_1} \left(\theta + \frac{\beta_1}{\beta_2} (\theta + T(\beta_2)) \right) \\ &= \cdots = \frac{S}{\beta_1} \left(\theta + \frac{\beta_1}{\beta_2} \left(\theta + \frac{\beta_2}{\beta_3} \right. \right. \\ &\quad \left. \left. \times \left(\theta + \cdots + \frac{\beta_{n-1}}{\beta_n} (\theta + \beta_n) \cdots \right) \right) \right) \\ &= S \cdot \theta \cdot \left(\frac{1}{\theta} + \frac{1}{\beta_1} + \frac{1}{\beta_2} + \cdots + \frac{1}{\beta_n} \right) \end{aligned} \tag{14}$$

The total storage for all subroutines with their entry overhead (disregarding base storage costs, M) is

$$W = \frac{S}{\beta_1} + \left(\theta + \frac{\beta_1}{\beta_2}\right) + \cdots + \left(\theta + \frac{\beta_{n-1}}{\beta_n}\right) + (\theta + \beta_n) = n \cdot \theta + \sum_{j=0}^{n} \frac{\beta_j}{\beta_{j+1}} \tag{15}$$

The task cost in this implementation is, as before, $W \cdot T(S)$. For fixed n, the minimum cost can be determined by treating the $\{\beta_i\}$ as continuous variables, and finding the zero(s) of the derivatives of cost with respect to each β_i.

$$0 = \beta_i \cdot \frac{\partial \ell n\,(c)}{\partial \beta_i} = \frac{\dfrac{\beta_i}{\beta_{i+1}} - \dfrac{\beta_{i-1}}{\beta_i}}{n\theta + \sum_{j=0}^{n} \dfrac{\beta_j}{\beta_{j+1}}} - \frac{\dfrac{1}{\beta_i}}{\dfrac{1}{\theta} + \sum_{j=1}^{n} \dfrac{1}{\beta_j}}, \quad \forall_i \tag{16}$$

Direct analytical solution of Eq. (16) appears hopeless, so the reasonable tactic is to search for upper and lower bounds to the $\{\beta_j\}$.

For an assumed fixed n, the lower bound is constructed quite simply as the solution to

$$0 = \frac{\beta_i}{\beta_{i+1}} - \frac{\beta_{i-1}}{\beta_i}, \quad \forall_i \tag{17}$$

Any solution to Eq. (17) makes the right hand side of Eq. (16) negative for all i (since an always negative term was dropped from Eq. (16) to produce Eq. (17)). This in turn implies that minute increases in any or all of the β_i will decrease task cost, and hence that solutions to Eq. (17) are everywhere below the optimum $\{\beta_j\}$. Applying to Eq. (17) the boundary conditions that $\beta_0 = S$, and $\beta_{n+1} = 1$ produces the solution

$$\underline{\beta}_i = S^{(n+1-i)/(n+1)}, \quad \forall_i \tag{18}$$

Rearranging Eq. (16) into the form of Eq. (17) plus an error term is a first step in developing an upper bound solution:

$$0 = \frac{\beta_i}{\beta_{i+1}} - \frac{\beta_{i-1}}{\beta_i} - \frac{1}{\beta_i}\left\{\frac{n\theta + \sum_{j=0}^{n} \beta_j/\beta_{j+1}}{1/\theta + \sum_{j=1}^{n} 1/\beta_j}\right\} \tag{19}$$

Substitution of Eq. (18) into the bracketed expression of Eq. (19) produces $n\theta S^{1/(n+1)}$. Thus at $\underline{\beta}_{n-j}$, Eq. (19) is approximately $n \cdot \theta \cdot S^{-j/(n+1)}$. For large S, moderate n, most j, this is much less than $\underline{\beta}_{j-1}/\beta_j = S^{1/(n+1)}$. Thus Eq. (17) is almost identical to Eq. (19) for most j, and it is only for j near n that significant error is introduced.

The subroutine depth, the value of n, for which task cost is minimum is yet to be determined. We restrict the $\{\beta_i\}$, the subroutine sizes, to be those of Eq. (18). For convenience, let $\beta \equiv \underline{\beta}_n = S^{1/(n+1)}$. Inserting Eq. (18) into Eq. (15) and Eq. (14), the storage cost W, and task time T are given by

$$W = n\theta + (n+1)\beta$$
$$T = S\theta \cdot \left(\frac{1}{\theta} + \frac{1-\beta^{-n}}{\beta - 1}\right) \tag{20}$$

The total task cost is $C = W \cdot T$. The expression can be simplified somewhat by assuming $n >> 1$, $\beta^{-n} << 1$:

$$C = n \cdot S \cdot (\theta + \beta) \cdot \left(1 + \frac{\theta}{\beta - 1}\right) \tag{21}$$

The value of n that minimizes C is found by setting its first derivative to 0:

$$0 = n \cdot c \cdot \frac{dC}{dn} = 1 - \beta\, \ell n \beta \left[\frac{1}{\theta + \beta} - \frac{\theta/(\beta - 1)}{(\beta - 1) + \theta}\right] \tag{22}$$

Equation (22) could be solved directly numerically to relate β, the smallest subroutine size, to the overhead, θ, but an approximate analytic answer may be obtained by assuming $\beta - 1 \approx \beta$:

$$0 = 1 - \ell n \beta \left[\frac{\beta - \theta}{\beta + \theta}\right] \tag{23}$$

Eq. (23) is made tractable by the approximation

$$2\left\{\frac{\beta - \theta}{\beta + \theta}\right\} \approx \ell n \beta - \ell n \theta$$

which is valid to within a few percent for $\theta > 10$. With this approximation, Eq. (23) becomes a quadratic in $\ell n\,(\beta)$ with the solution

$$\ell n\,(\beta) = \ell n\,(\theta) \cdot \frac{1 + \left(\dfrac{1+8}{(\ell n \theta)^2}\right)^{1/2}}{2}$$
$$n = \frac{\ell n S}{\dfrac{\ell n \theta}{2} + \left[\left(\dfrac{\ell n \theta}{2}\right)^2 + 2\right]^{1/2}} \tag{24}$$

For large θ, β is approximately equal to θ, and task cost is given by

$$C \approx 4\theta S \ell n\,(S)/\ell n\,(\theta) \tag{25}$$

The overhead θ sets a lower bound to the smallest subroutine size, and hence places an upper limit to n. As the $\{\beta_j\}$ found in Eq. (18) represent a lower bound component-wise to the true minimizing $\{\beta_j\}$, we expect that the n, which minimizes cost at the true $\{\beta_j\}$, will be greater than the n found in Eq. (24).

To find an upper bound solution for the $\{\beta_i\}$, we must determine conditions under which $\{\partial C/\partial\beta_i\}$ is positive for all i. Equation (16) can be rearranged into the form

$$H^+(\beta)\frac{\partial C}{\partial \beta_i} = \beta_i^2 - \beta_{i+1}\cdot\left(\beta_{i-1} + \frac{n\theta + \sum_0^n \beta_j/\beta_{j+1}}{1/\theta + \sum_1^n 1/\beta_j}\right) \tag{26}$$

where $H^+(\beta)$ is an always positive function of $\{\beta\}$. Let us denote as $g(\theta, n, \beta)$ the fraction in parentheses in Eq. (26), and as g^+, an upper bound to $g(\theta, n, \beta)$. The lower-bound solution to $\{\beta_i\}$ is given in Eq. (18). We construct a tentative upper-bound solution from it as

$$\beta_i^0 = \left(\frac{n+1-i}{S^{n+1}}\right)^{\eta} S^{1-\eta} \tag{27}$$

where η is a small constant yet to be determined. A second tentative upper bound β_i^1 is constructed from the first by solving sequentially from $i = n$ down:

$$\begin{aligned} \beta_{n+1}^1 &= 1 \\ \beta_i^1 &= [\beta_{i+1}^1\cdot(\beta_{i-1}^0 + g^+)]^{1/2} \end{aligned} \tag{28}$$

With this solution, η is adjusted (if possible) so that

$$\beta_i^0 \geqq \beta_i^1, \qquad \text{all } i \tag{29}$$

This restriction implies that

$$\begin{aligned} \beta_i^1 &\geqq [\beta_{i+1}^1\cdot(\beta_{i-1}^1 + g^+)]^{1/2} \\ \beta_i^1 &\geqq [\beta_{i+1}^1\cdot(\beta_{i-1}^1 + g(\theta, n, \beta^1))]^{1/2} \end{aligned} \tag{30}$$

which in turn implies that

$$\frac{\partial C}{\partial \beta_i} \geqq 0, \qquad \forall_i, \qquad \text{at } \beta_i^1 \tag{31}$$

That is, $\{\beta_i^1\}$ is an upper-bound solution to $\{\beta_i\}$, but as $\beta_i^0 \geqq \beta_i^1$ for all i, $\{\beta_i^0\}$ is also an upper-bound solution for the selected η. It remains to determine η.

We expand the solution for β_i^1 by substituting Eq. (27) into Eq. (28) and iterating downward:

$$\beta_{n-K}^1 = \prod_{j=1}^{K+1}\left[S^{\eta(K+3-j)/(n+1)} + g_+\right]^{2^{-j}}$$

$$\beta_{n-K}^1 = S^{\eta(K+1)/(n+1)}\cdot S^{(1-\eta)(1-2^{-K-1})} \times \prod_{j=1}^{K+1}\left[1 + \frac{g_+}{S^{\eta(K+3-j)/(n+1)}\cdot S^{1-\eta}}\right]^{2^{-j}} \tag{32}$$

The product in Eq. (32) can be converted to a sum by the use of logarithms, and a summable upper bound to the sum of logarithms is generated by the use of $\ell n\,(1+X) \leqq X$, for all X.

$$\beta_{n-K}^1 \leqq S^{\eta(K+1)/(n+1)}\cdot S^{(1-\eta)(1-2^{-K-1})} \times \exp\left\{\frac{g_+}{S^{1-\eta}}\sum_{j=1}^{K+1}\frac{1}{2^j S^{\eta(K+3-j)/(n+1)}}\right\} \tag{33}$$

The sum in Eq. (33) is a simple geometric progression. If the right hand side of Eq. (33) is forced to be below $\beta_{n-K}^0 = S^{\eta(K+1)/(n+1)}\cdot S^{1-\eta}$, then $\beta_{n-K}^1 \leqq \beta_{n-K}^0$, as desired. For convenience, denote $\beta \triangleq \beta_n^{\eta} = S^{\eta/(n+1)}$. The bound conconstraint becomes

$$S^{(1-\eta)(-2^{-K-1})}\exp\left\{\frac{g_+}{S^{1-\eta}}\beta^{-(K+3)}\frac{\beta^{K+2}-\beta}{\beta-1}\right\} \leqq 1 \tag{34}$$

or

$$g_+\cdot\frac{1-\left(\frac{2}{\beta}\right)^{K+1}}{1-\left(\frac{2}{\beta}\right)}\cdot\beta^{-2} \leqq S^{1-\eta}\,\ell n S^{1-\eta}, \qquad \forall_K \tag{35}$$

If we assume $\beta > 2$, then the inequality of Eq. (35) is assured whenever

$$g_+ \leqq \beta(\beta-2)\,S^{1-\eta}\,\ell n S^{1-\eta} \tag{36}$$

A value for g_+ can be conveniently derived by manipulating $g(\theta, n, \beta^0)$: $g_+ = \theta(n\theta + n\beta + S^{1-\eta})$, thus

$$\theta(n\theta + n\beta + S^{1-\eta}) \leqq \beta(\beta-2)\,S^{1-\eta}\,\ell n S^{1-\eta} \tag{37}$$

Let us assume an n that varies with S as $n = \lambda\,\ell n S$, for λ constant. This is consistent with past findings Eq. (24) at

the lower bound. If n grows more rapidly than $\lambda \ell nS$, then task cost will grow more rapidly than $S \cdot \ell nS$, and it will be seen that this is achievable at $n = \lambda \ell nS$. Assuming η is approximately 1, substitution of $n = \lambda \ell nS$ into the inequality Eq. (37) produces

$$\theta \cdot (\lambda \ell nS\,[\theta + e^{1/\lambda}] + S^{1-\eta}) \leq (e^{2/\lambda} - 2e^{1/\lambda}) \cdot S^{1-\eta}\,\ell nS^{1-\eta} \tag{38}$$

It can be shown that $S^{1-\eta}$ must grow at least as fast as $\mu \cdot \ell nS$ (for μ constant) to satisfy Eq. (38).

$$\theta \cdot \frac{\lambda}{\mu}\,[\theta + e^{1/\lambda}] + 1\Big) \leq (e^{2/\lambda} - 2e^{1/\lambda})\,\ell n\,(\mu\,\ell nS) \tag{39}$$

Equation (39) is satisfied for large S for any arbitrarily small constant μ since the right hand side is increasing with S. For $S^{1-\eta} = \mu\,\ell nS$, η converges to 1 for large S.

Using these forms for n and $S^{1-\eta}$, the task time and storage are given by

$$\left.\begin{aligned} T &= S \cdot \theta\left(\frac{1}{\theta} + \frac{1}{\mu\,\ell nS}\,\frac{1}{e^{1/\lambda} - 1}\right) \\ W &= \ell n\,(S)\,(\lambda\,[\theta + e^{1/\lambda}] + \mu e^{1/\lambda}) \end{aligned}\right\} \tag{40}$$

We are particularly interested in behavior at large S, so the second term of T may be ignored, as it converges to 0 for large S. The last term of W can also be ignored as μ can be an arbitrarily small constant, negligible with respect to λ. Thus

$$\begin{aligned} T &\approx S \\ W &\approx \ell n\,(S) \cdot \lambda\,[\theta + e^{1/\lambda}] \end{aligned} \tag{41}$$

The minimum of $C = T \cdot W$ is at that λ that minimizes storage, i.e., at

$$\theta = \beta\,(\ell n\beta - 1), \qquad \text{where } \beta = e^{1/\lambda} \tag{42}$$

The minimum cost is thus

$$C = S \cdot \ell n\,(S) \cdot \beta \tag{43}$$

and

$$C \gtrsim S \cdot \ell n\,(S) \cdot \frac{\theta}{\ell n\theta - 1} \tag{44}$$

The value of β defined from Eq. (42) is below the lower-bound β for all values of θ. Hence the value of n which corresponds to Eq. (42) is larger than that of Eq. (24), and it may be argued that this n is an upper bound to the true n.

From the form of the upper- and lower-bound solutions, it may reasonably be inferred that the subroutine depth n for minimum task cost is given by $n = \lambda\,\ell n\,S$, for λ some small constant which makes $S^{1/n} \approx \theta$, and that minimum task cost is equal to $\nu \cdot S \cdot \ell n\,S \cdot \theta/\ell n\theta$ where ν is a small constant on the order of 1.

V. Interpretation

One can perform almost the same sequence of analysis for execution with an interpreter as for multiply-nested subroutines. The interpreter is another tool with which a software system designer can reduce storage at the expense of increased execution time. The interpreter requires some fixed minimum overhead in storage to hold the interpretive mechanism itself, and a smaller amount of storage associated with subroutine linkage via the interpreter. There is a time overhead associated with interpreted subroutine linkage which is greater than the corresponding storage overhead, and there is a time overhead associated with each interpreted step. To counterbalance these effects, the storage required for an interpreted step is reduced by a factor of 2 to 10 (or more) from what it would be without the interpreter. Although the specific parameters are different from those of the multilevel subroutines, the conclusion reached is certain to be quantitatively the same as just presented for subroutines: that the decision to fold the program is a good one provided the work performed in the lowest level routines is roughly equal to the overhead work encountered in getting there.

There are a wide variety of other options that have not and will not be covered here. Both program code and data may be stored in a secondary memory, and called to main storage when needed, instead of being assumed to all reside in main storage as was done here. This call for secondary memory data may be either program controlled, or under system control as part of a virtual memory structure. In the latter case, both processor and swapping scheduling algorithms affect the task cost, as does the installation's pricing policies. Another piece of this massive "jigsaw puzzle" called computational efficiency is the effect of memories and processors with a variable cost that is some function of the innate device speed. These are some of the open questions.

VI. Implications for Reality

The analysis just presented has attempted to reveal the efficiency implications of some conventional software structures. The actual problems solved are abstractions of the real programming questions of how much a program should be folded by loop and subroutine control structures. Such questions are of direct interest to two types of people: to system designers who can build or buy exactly the right amount of hardware to do their job, and to users of large-scale, time-sharing, multiprogramming systems where one pays only for resources used. Our departure from reality consists of the drastic assumption that the programs can be folded arbitrarily, i.e. without regard to content. While this is clearly false, and hence degrades the quantitative conclusions, the qualitative conclusion should remain valid, namely: whenever the storage for a task is dominated by program rather than data, the decision to fold that program to reduce its storage, by any available means, is a good one provided that the work performed in the lowest level routines is roughly equal to the overhead involved in the control structures of the higher levels.

The Structured Programming and "levels of abstraction" techniques (Ref. 2) is currently growing in popularity for the development of complex software systems that are transparent and readily understandable. The technique leads naturally to extensive use of multiple level subroutines. Are such programs efficient? The analysis presented above says *yes*.

References

1. Savage, J. E., and Harper, L. H., "Contributions to a Mathematical Theory of Complexity," in *The DSN Progress Report*, Technical Report 32-1526, Vol. V, Jet Propulsion Laboratory, Pasadena, Calif., Oct. 15, 1971.
2. Baker, F. T., "System Quality through Structured Programming," *AFIPS Conference Proceedings*, Vol. 41, Part I, pp. 339–343, 1972 Fall Joint Computer Conference, Los Angeles, Calif., Dec. 5–7, 1972.